Enhancing the scalability and load balancing of the parallel selected inversion algorithm via tree-based asynchronous communication

Mathias Jacquelin
Lawrence Berkeley National Laboratory
mjacquelin@lbl.gov

Lin Lin
University of California Berkeley
Lawrence Berkeley National Laboratory
linlin@math.berkeley.edu

Nathan Wichmann
Cray Inc.
wichmann@cray.com

Chao Yang
Lawrence Berkeley National Laboratory
cyang@lbl.gov

ABSTRACT

We develop a method for improving the parallel scalability of the recently developed parallel selected inversion algorithm [Jacquelin, Lin and Yang 2014], named `PSelInv`, on massively parallel distributed memory machines. In the `PSelInv` method, we compute selected elements of the inverse of a sparse matrix A that can be decomposed as $A = LU$, where L is lower triangular and U is upper triangular. Updating these selected elements of A^{-1} requires restricted collective communications among a subset of processors within each column or row communication group created by a block cyclic distribution of L and U. We describe how this type of restricted collective communication can be implemented by using asynchronous point-to-point MPI communication functions combined with a binary tree based data propagation scheme. Because multiple restricted collective communications may take place at the same time in the parallel selected inversion algorithm, we need to use a heuristic to prevent processors participating in multiple collective communications from receiving too many messages. This heuristic allows us to reduce communication load imbalance and improve the overall scalability of the selected inversion algorithm. For instance, when $6,400$ processors are used, we observe over 5x speedup for test matrices. It also mitigates the performance variability introduced by an inhomogeneous network topology.

Keywords

selected inversion, distributed memory parallel algorithm, asynchronous data communication, high performance computation, load balancing

1. INTRODUCTION

Collective communication such as broadcast and reduction is an ubiquitous type of communication used in many parallel programs. When such communication is required among all processors that belong to a communication group labeled by a communicator, one can use standard message passing interface (MPI) functions such as `MPI_Bcast` and `MPI_Reduce`.The MPI libraries available on most of high performance computers often provide highly efficient implementations of these functions. These implementations typically make use of a tree-based algorithm that minimizes the total communication volume and the number of messages.

However, in some applications, collective communication is required only among a subset of processors within a predefined communication group, and this subset of processors may change over time. One such application is the pole expansion and selected inversion method [2, 4, 5] that can be used to accelerate Kohn-Sham density functional theory [6] based electronic structure calculations. Because the current MPI standard does not support collective communication among an arbitrary subset of processors, one must resort to other mechanisms to accomplish such a communication task.

One possible solution is to determine all collective communication calls that will be needed in advance and the processors involved in each one of these calls, set up multiple communication groups, and use them whenever they are needed. However, the total number of communication groups needed (e.g., in the selected inversion algorithm) may exceed the capacity of the MPI libraries, which is typically around several thousands (currently 4,096 on Cray MPI for instance). Hence the approach of pre-allocating all communicators is not feasible for all applications.

Another approach is to create communication groups dynamically as they are needed, and release them when they are no longer needed. However, this approach typically incurs a significant amount of overhead that interferes with the asynchronous nature of the parallel selected inversion algorithm, and thus limits its parallel scalability on large scale distributed memory machines.

Yet another solution is to replace the collective communication altogether with point-to-point communications. Although this approach is plausible when each collective communication involves only a few processors distributed among a small network of processors, it quickly becomes inefficient when the number of processors involved in the communication becomes large. One main pitfall of this approach is that communication load is not well balanced among different processors. Such imbalance can severely impair the overall parallel performance. Furthermore, when executed on massively parallel machines that have a hierarchical and inhomogeneous network architecture, such an approach also introduces performance variability.

However, improvement can be made to reduce the overall communication cost if we orchestra the point-to-point communication in such a way that mimics the collective communication implemented in standard MPI libraries. That is, if we combine a tree-based algorithm with asynchronous point-to-point sends and receives, we can effectively construct dynamic collective communications among an arbitrary subset of processors.

In this paper, we demonstrate that the use of the third option can be quite effective. However, it needs to be implemented with care to accommodate a special feature of the selected inversion algorithm that allows several restricted collective communications to take place at the same time. We present a heuristic that prevent processors participating in multiple collective communications from receiving too many messages. We show that this heuristic is very effective in reducing the amount of load imbalance. It also reduces performance variability induced by an inhomogeneous network architecture.

Our paper is organized as follows. In the next section, we briefly describe the selected inversion algorithm and its parallel implementation. We point out the nature of collective communications required in the parallel implementation that calls for the implementation of customized broadcast and reduction operations built on top of asynchronous point-to-point communications. We discuss the construction of binary trees to propagate data among different processors and the heuristic for improving communication load balance. In section 4, we report the performance improvement achieved by this technique. In particular, we show that the implementation of dynamic collective communication allows the selected inversion method to scale efficiently beyond 4000 processors. The wall clock time can be reduced by more than a factor of three, and the overall variation in runtime is also reduced. This improvement enables us to use selected inversion based electronic structure calculation to over 100,000 cores with the pole expansion and selected inversion technique [2, 5].

2. SELECTED INVERSION

Let $A \in \mathbb{C}^{N\times N}$ be a non-singular sparse matrix. We use $A_{i,j}$ to denote the (i,j)-th entry of the matrix A, and $A_{i,*}$ and $A_{*,j}$ to denote the i-th row and the j-th column of A, respectively. We are interested in computing *selected elements* of A^{-1}, defined as

$$\{(A^{-1})_{i,j} | \text{ for } 1 \le i,j \le N, \text{ such that } A_{i,j} \neq 0\}. \quad (1)$$

Sometimes, we only need to compute a subset of these selected elements, for example, the diagonal elements of A^{-1}. The most straightforward way to obtain these selected elements of A^{-1} is to compute the full inverse of A and then extract the selected elements. But this is often prohibitively expensive in practice. If a sparse LU factorization of A is available (or LDL^T factorization if A is symmetric) , a more efficient way to achieve this goal is to use an algorithm that makes efficient use of the sparse L and U factors of A. In such an algorithm, which we call selected inversion (*SelInv*) some additional elements of A^{-1} may need to be computed. However, the overall set of nonzero elements that need to be computed often remains a small percentage of all elements of A^{-1} due to the sparsity structure of A.

The selected inversion algorithm and its variants have been discussed in a number of publications [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 4, 3]. We review the basic ingredients of this algorithm in section 2.1 and describe the recently developed parallel algorithm in section 2.2.

2.1 Sequential algorithm

The selected inversion algorithm can be derived as follows. Given a 2-by-2 block partitioning of matrix A of the form

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, \quad (2)$$

where $A_{1,1}$ is a scalar entry of A. $A_{1,1}$ can be expressed as a product of two scalars $L_{1,1}$ and $U_{1,1}$. In particular, we can pick $L_{1,1} = 1$ and $U_{1,1} = A_{1,1}$. Then

$$A = \begin{pmatrix} L_{1,1} & 0 \\ L_{2,1} & I \end{pmatrix} \begin{pmatrix} U_{1,1} & U_{1,2} \\ 0 & S_{2,2} \end{pmatrix} \quad (3)$$

where

$$L_{2,1} = A_{2,1}(U_{1,1})^{-1}, \quad U_{1,2} = (L_{1,1})^{-1}A_{1,2}. \quad (4)$$

The L and U factors are usually directly accessible in a standard LU factorization, and

$$S_{2,2} = A_{2,2} - L_{2,1}U_{1,2} \quad (5)$$

is the Schur complement. Using the decomposition given by Eq. (3), we can express A^{-1} as

$$A^{-1} = \begin{pmatrix} (U_{1,1})^{-1}(L_{1,1})^{-1} & \\ +(U_{1,1})^{-1}U_{1,2}S_{2,2}^{-1}L_{2,1}(L_{1,1})^{-1} & -(U_{1,1})^{-1}U_{1,2}S_{2,2}^{-1} \\ -S_{2,2}^{-1}L_{2,1}(L_{1,1})^{-1} & S_{2,2}^{-1} \end{pmatrix}. \quad (6)$$

Since $S_{2,2}$ is the same as S here, without ambiguity $S_{2,2}^{-1} \equiv (S^{-1})_{2,2}$ can be used. To simplify the notation, we define the normalized LU factors as

$$\begin{array}{ll} \hat{L}_{1,1} = L_{1,1}, & \hat{U}_{1,1} = U_{1,1}, \\ \hat{L}_{2,1} = L_{2,1}(L_{1,1})^{-1}, & \hat{U}_{1,2} = (U_{1,1})^{-1}U_{1,2}, \end{array} \quad (7)$$

and Eq. (6) can be equivalently given by

$$A^{-1} = \begin{pmatrix} (\hat{U}_{1,1})^{-1}(\hat{L}_{1,1})^{-1} + \hat{U}_{1,2}S_{2,2}^{-1}\hat{L}_{2,1} & -\hat{U}_{1,2}S_{2,2}^{-1} \\ -S_{2,2}^{-1}\hat{L}_{2,1} & S_{2,2}^{-1} \end{pmatrix}. \tag{8}$$

Let us denote by $\mathcal{C}$ the set of indices

$$\{i|\,(L_{2,1})_i \neq 0\} \cup \{j|\,(U_{1,2})_j \neq 0\}, \tag{9}$$

and assume $S_{2,2}^{-1}$ has already been computed. From Eq. (8) it can be readily observed that in order to compute the selected elements of $\left(A_{2,1}^{-1}\right)_i \equiv -\left(S_{2,2}^{-1}\hat{L}_{2,1}\right)_i$ for $i \in \mathcal{C}$, we only need the entries

$$\left\{\left(S_{2,2}^{-1}\right)_{i,j} | i \in \mathcal{C}, j \in \mathcal{C}\right\}. \tag{10}$$

The same set of entries of $S_{2,2}^{-1}$ are required to compute selected entries of $A_{1,2}^{-1} \equiv -\hat{U}_{1,2}S_{2,2}^{-1}$. No additional entries of $S_{2,2}^{-1}$ are needed to complete the computation of $A_{1,1}^{-1}$, which involves the matrix product of selected entries of $\hat{U}_{1,2}$ and $A_{2,1}^{-1}$. This procedure can be repeated recursively to compute selected elements of $S_{2,2}^{-1}$ until $S_{2,2}$ is a scalar of size 1. A pseudo-code for demonstrating this column-based selected inversion algorithm for symmetric matrix is given in [4].

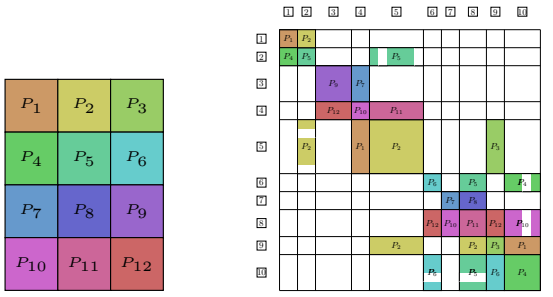
Algorithm 1: Selected inversion algorithm based on *LU* factorization.

Input: (1) The supernode partition of columns of A: $\{1, 2, ..., \mathcal{N}\}$
(2) A supernodal LU factorization of A with (unnormalized) LU factors L and U.

Output: Selected elements of A^{-1}, i.e. $A_{\mathcal{I},\mathcal{J}}^{-1}$ such that $L_{\mathcal{I},\mathcal{J}}$ is not an empty block.

for $\mathcal{K} = \mathcal{N}, \mathcal{N}-1, ..., 1$ **do**
1 Find the collection of indices $\mathcal{C} = \{\mathcal{I} \mid \mathcal{I} > \mathcal{K}, L_{\mathcal{I},\mathcal{K}} \text{ is a nonzero block}\} \cup \{\mathcal{J} \mid \mathcal{J} > \mathcal{K}, U_{\mathcal{K},\mathcal{J}} \text{ is a nonzero block}\}$
2 $\hat{L}_{\mathcal{C},\mathcal{K}} \leftarrow L_{\mathcal{C},\mathcal{K}}(L_{\mathcal{K},\mathcal{K}})^{-1}, \hat{U}_{\mathcal{K},\mathcal{C}} \leftarrow (U_{\mathcal{K},\mathcal{K}})^{-1}U_{\mathcal{K},\mathcal{C}}$
end
for $\mathcal{K} = \mathcal{N}, \mathcal{N}-1, ..., 1$ **do**
Find the collection of indices $\mathcal{C} = \{\mathcal{I} \mid \mathcal{I} > \mathcal{K}, L_{\mathcal{I},\mathcal{K}} \text{ is a nonzero block}\} \cup \{\mathcal{J} \mid \mathcal{J} > \mathcal{K}, U_{\mathcal{K},\mathcal{J}} \text{ is a nonzero block}\}$
3 Calculate $A_{\mathcal{C},\mathcal{K}}^{-1} \leftarrow -A_{\mathcal{C},\mathcal{C}}^{-1}\hat{L}_{\mathcal{C},\mathcal{K}}$
4 Calculate $A_{\mathcal{K},\mathcal{K}}^{-1} \leftarrow U_{\mathcal{K},\mathcal{K}}^{-1}L_{\mathcal{K},\mathcal{K}}^{-1} - \hat{U}_{\mathcal{K},\mathcal{C}}A_{\mathcal{C},\mathcal{K}}^{-1}$
5 Calculate $A_{\mathcal{K},\mathcal{C}}^{-1} \leftarrow -\hat{U}_{\mathcal{K},\mathcal{C}}A_{\mathcal{C},\mathcal{C}}^{-1}$
end

In practice, a column-based sparse factorization and selected inversion algorithm may not be efficient due to the lack of level 3 BLAS operations. For a sparse matrix A, the columns of A and the L factor can be partitioned into supernodes. A supernode is a maximal set of contiguous columns $\mathcal{J} = \{j, j+1, \ldots, j+s\}$ of the L factor that have the same nonzero structure below the $(j+s)$-th row, and the lower triangular part of $L_{\mathcal{J},\mathcal{J}}$ is dense. This definition can be relaxed to limit the maximal number of columns in a supernode (i.e. sets are not necessarily maximal). With slight abuse of notation, both a supernode index and the set of column indices associated with a supernode are denoted by uppercase script letters such as $\mathcal{I}, \mathcal{J}, \mathcal{K}$ *etc.*. $A_{\mathcal{I},*}$ and $A_{*,\mathcal{J}}$ are used to denote the $\mathcal{I}$-th block row and the $\mathcal{J}$-th block column of A, respectively. $A_{\mathcal{I},\mathcal{J}}^{-1}$ denotes the $(\mathcal{I},\mathcal{J})$-th block of the matrix A^{-1}, i.e. $A_{\mathcal{I},\mathcal{J}}^{-1} \equiv (A^{-1})_{\mathcal{I},\mathcal{J}}$. When the block $A_{\mathcal{I},\mathcal{J}}$ itself is invertible, its inverse is denoted by $(A_{\mathcal{I},\mathcal{J}})^{-1}$ to distinguish from $A_{\mathcal{I},\mathcal{J}}^{-1}$.



(a) A 4-by-3 example of a 2D processor grid

(b) 2D block cyclic distribution of PSelInv sparse matrix data structure on a 4-by-3 processor grid

Figure 1: Data layout of the internal sparse matrix data structure used by PSelInv.

Using the supernode notation, a pseudo-code for the selected inversion algorithm is given in Algorithm 1.

2.2 Parallel selected inversion algorithm

In this section we briefly discuss the parallel implementation of the selected inversion algorithm, called PSelInv, on distributed memory parallel machines. More details of the implementation for symmetric matrices can be found in [17].

PSelInv uses the same 2D block cyclic distribution scheme employed by SuperLU_DIST [18] to partition and distribute both the L factor and the selected elements of A^{-1} to be computed. Before the factorization, columns of A, L and U are partitioned into supernodes of various sizes. This partition is applied to the rows of the input matrix to create a 2D block partition. These blocks are cyclically mapped onto processors that are arranged in a virtual Pr-by-Pc 2D grid. The mapping itself does not take the sparsity of the matrix into account, however only non-zero elements are actually stored. As an example, a 4-by-3 grid of processors is depicted in Figure 1(a). The mapping of the 2D supernode partition of the matrix on the 2D processor grid is depicted in Figure 1(b). Each supernodal block column of L is distributed among processors that belong to a column of the processor grid. Each processor may own multiple matrix blocks. For instance, nonzero rows in the second supernode are owned by processors P_2 and P_5.

We execute the first loop of Algorithm 1 in a separate pass, since the data communication required in this step is relatively simple. The processor that owns the block $L_{\mathcal{K},\mathcal{K}}$ broadcasts it to all processors that own nonzero blocks $L_{\mathcal{I},\mathcal{K}}$ in the supernode $\mathcal{K}$ within the same processor column. Each of those processors performs the triangular solve $\hat{L}_{\mathcal{I},\mathcal{K}} \equiv L_{\mathcal{I},\mathcal{K}}(L_{\mathcal{K},\mathcal{K}})^{-1}$ for each nonzero block contained in the set $\mathcal{C}$ defined in step 1 of the algorithm. Because $L_{\mathcal{I},\mathcal{K}}$ is not used in the subsequent steps of selected inversion, it is overwritten by $\hat{L}_{\mathcal{I},\mathcal{K}}$. Since communication is limited to a processor

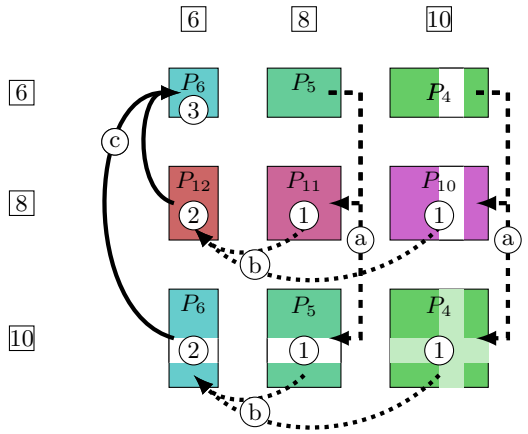


Figure 2: Task parallelism and communication pattern for the supernode 6. There are 6 steps: (a) broadcast $\hat{L}$, (1) compute $A^{-1}\hat{L}$, (b) reduce $A^{-1}\hat{L}$, (2) compute $\hat{L}^T A^{-1}\hat{L}$, (c) reduce $\hat{L}^T A^{-1}\hat{L}$ and (3) update A^{-1}.

column group only, multiple supernodes can be processed at the same time.

A more complicated communication pattern, which also turns out to be the most time consuming step in terms of data communication, is required to complete step 3 in parallel. $A^{-1}_{\mathcal{C},\mathcal{C}}$ and $\hat{L}_{\mathcal{C},\mathcal{K}}$ are generally owned by different processor sets. In `PSelInv`, we choose to send the $\hat{L}_{\mathcal{C},\mathcal{K}}$ matrix blocks to processors owning the *matching* blocks of $A^{-1}_{\mathcal{C},\mathcal{C}}$ to perform the matrix-matrix multiplication, i.e. a particular matrix block $\hat{L}_{\mathcal{I},\mathcal{K}}$ needs to be communicated to all processors within the same column group of processors among which $A^{-1}_{\mathcal{C},\mathcal{I}}$ is distributed.

However, since the processor owning $\hat{L}_{\mathcal{I},\mathcal{K}}$ is generally not in the same processor row/column group that owns $A^{-1}_{\mathcal{C},\mathcal{I}}$, the communication cannot be performed by using a simple broadcast procedure. We briefly describe the communication pattern for this step here, since most of the communication cost is spent on this step. In `PSelInv`, we use point-to-point MPI sends that originate from the processor owning $\hat{L}_{\mathcal{I},\mathcal{K}}$ to the group of processors holding $A^{-1}_{\mathcal{C},\mathcal{I}}$.

For symmetric matrices, as soon as $\hat{L}_{\mathcal{I},\mathcal{K}}$ becomes available, as illustrated above, it is sent to the processor owning $\hat{U}_{\mathcal{K},\mathcal{I}}$, which is then overwritten by $\hat{L}^T_{\mathcal{I},\mathcal{K}}$. Once $\hat{L}_{\mathcal{I},\mathcal{K}}$ has been sent the processor mapped to the upper triangular part of the matrix, step 3 of Algorithm 1 can be performed. $\hat{U}_{\mathcal{K},\mathcal{I}} = \hat{L}^T_{\mathcal{I},\mathcal{K}}$ is first sent to all processors within the same processor column that owns $\hat{U}_{\mathcal{K},\mathcal{I}}$. The matrix-matrix product $A^{-1}_{\mathcal{J},\mathcal{I}}\hat{L}_{\mathcal{I},\mathcal{K}}$ is then performed locally on each processor owning $A^{-1}_{\mathcal{J},\mathcal{I}}$ using BLAS3 kernel. Then, contributions $A^{-1}_{\mathcal{J},\mathcal{I}}\hat{L}_{\mathcal{I},\mathcal{K}}$ are reduced within each processor rows owning $\hat{L}_{\mathcal{J},\mathcal{K}}$. The $A^{-1}_{\mathcal{J},\mathcal{K}}$ block in step 3 of Algorithm 1 is now fully computed.

Figure 2 illustrates how this step is completed for supernode $\mathcal{K} = 6$. We use circled letters (a), (b), (c) to label communication events, and circled numbers (1), (2), (3) to label computational events. $\hat{U}_{6,8} = \hat{L}^T_{8,6}$ is sent by P_5 to all processors within the processor column. This group include both P_5 and P_{11}. Similarly $\hat{L}_{10,6}$ is broadcast from P_4 to all other processors within the processor column. Local GEMMs are then performed on P_{11}, P_{10}, P_4 and P_5 simultaneously, before being reduced onto P_{12} and P_5 within their respective processor row. After this step, $A^{-1}_{8,6}$ and $A^{-1}_{10,6}$ become available on P_{12} and P_6 respectively.

In [17], we pointed out that an additional coarse-grained level of parallelism exists in the second loop of Algorithm 1. Different loop iterates can be executed simultaneously if 1) there is no data dependency among these iterates; 2) there is no overlap among the processors that own data blocks belonging to these loop iterates.

The absence of data dependency among different loop iterates results from the sparsity structure of A and its L and U factors, and can be exposed by the elimination tree [19] associated with a sparse LU factorization. Although two supernodes belonging to two different branches of the elimination tree would need to communicate with their common ancestors in the selected inversion algorithm, these communications do not have to take place at the same time. Thus, it is still possible to process these two supernodes simultaneously. However, due to the 2D cyclic distribution of the supernodes, it is possible that some of the matrix blocks belonging to two independent supernodes are owned by the same processor. In that case, full parallelism cannot be achieved between the two supernodes.

In `PSelInv`, we do not explicitly use the MPI_Barrier function for synchronization. The synchronization is only imposed through data dependencies. As a result, tasks associated with different supernodes can be executed concurrently if these supernodes are on different critical paths of the elimination tree, and if there is no overlap among processors mapped to these critical paths. In this sense, the asynchronous task formulation tries to achieve two goals: **pipelining** computations and **overlapping** communication with computations.

3. COLLECTIVE COMMUNICATION IN PARALLEL SELECTED INVERSION

We can clearly see that the parallelization strategy we presented in section 2.2 involves a fair amount of data communication. Most of this communication occurs in the second loop of Algorithm 1, and in particular, step 5 of the algorithm. Therefore, the performance of our parallel implementation of the selected inversion algorithm depends critically on how $\hat{L}_{\mathcal{I},\mathcal{K}}$ is sent to the matching blocks of $A^{-1}_{\mathcal{C},\mathcal{C}}$ and how local products $A^{-1}_{\mathcal{J},\mathcal{I}}\hat{L}_{\mathcal{I},\mathcal{K}}$ are reduced to the processor that owns the $\hat{L}_{\mathcal{J},\mathcal{K}}$ block in supernode $\mathcal{K}$. These data communications are **collective** in nature. However, they should be carefully treated in the sense that each **broadcast** operation labeled by (a) (or **reduction** operation labeled by (b)) in Figure 2 involves only a subset of processors within a column or row processor group defined within a virtual 2D processor grid shown in Figure 1(b). We will call this type of collective communication *restricted collective communication*. Furthermore, in `PSelInv`, the subset of processors involved in restricted collective communications varies

when different supernodes are processed due to the general sparsity structure of L and U.

As we indicated in the introduction, one way to implement these collective communication is to construct all possible communication groups, each containing a subset of processors involved in each **broadcast** and **reduction** operation respectively, in advance, and use `MPI_Bcast` and `MPI_Reduce` functions available in standard MPI libraries to perform these collective communications. However, for large problems, the number of communication groups required often exceeds what most MPI libraries can provide. Besides the overhead for creating a large number of MPI communicators is non-negligible. Thus, this approach is not viable.

Although it is possible to create these communication groups dynamically, frequent creation and release of communication groups tends to result in an excessive amount of overhead. Even if this overhead is negligible, using `MPI_Bcast` and `MPI_Reduce` is still not optimal because the *collective* and *blocking* nature of these functions reduces the opportunity to exploit the loop level concurrency available among different supernodes. The subset of ranks involved in one broadcast may be different from the subset involved in another broadcast, but it is highly likely that the at least some of the ranks in one broadcast will also be in another broadcast. Consequently, the broadcast of one block cannot proceed until the previous broadcast completes, making the pipelining of updates of different supernodes in an asynchronous fashion more difficult to achieve. Ideally, we would like to have a set of light-weight asynchronous **broadcast** and **reduction** functions that can be dynamically created with very little overhead.

We also note that the group of processors involved in each collective communication is determined once the L, U factors and the 2D processor mapping is given, and therefore no further communication is needed to set up the tree once the list of processors is known. With such a list, the tree structure can be created dynamically with very small overhead. The buffer arrays for performing the selected inversion is also created dynamically. Such functions are currently not available in standard MPI libraries. Therefore, we decided to implement this type of restricted collective communication through the use of point-to-point asynchronous communication functions such as `MPI_Isend` and `MPI_Irecv`.

In the implementation we presented in [17], we simply issue multiple `MPI_Isend`'s to send $\hat{L}_{\mathcal{I},\mathcal{K}}$ from one processor to other processors that own different matching blocks of $A^{-1}_{\mathcal{C},\mathcal{C}}$. Similarly, multiple `MPI_Irecv`'s are issued to accumulate the products $A^{-1}_{\mathcal{J},\mathcal{I}}\hat{L}_{\mathcal{J},\mathcal{K}}$ on the processor that owns $A^{-1}_{\mathcal{J},\mathcal{K}}$. By using this simple strategy, we were able to perform parallel selected inversion efficiently on $256 \sim 1,024$ processors depending on the sparsity pattern and the size of a matrix. However, when a larger number of processors are used, the performance of parallel selected inversion quickly deteriorates. The performance profile we measured indicated that communication cost became the dominant cost in those cases. For instance, for the DG_PNF14000 matrix in section 4, when a relatively small number of processors $P = 256$ is used, the communication cost is 27%, and the time spend on the computation, mainly the matrix-matrix multiplication (GEMM) routine is 73%. When a large number of processors $P = 4,096$ is used, the communication cost is 89%, and the time spend on the computation, mainly the matrix-matrix multiplication (GEMM) routine is only 11%.

A closer look at the communication profile reveals that the increase of communication cost is partly caused by a large variation in communication volumes consumed by different processors.

The communication imbalance is exacerbated by the inhomogeneity of the communication bandwidth and latency among different nodes and processors. Figure 8(a) shows that as the number of processors increases, the amount of **run time variation** also increases when we ran the same executable and input matrices multiple times. Since `PSelInv` is a deterministic algorithm, the run time variation is likely caused by variation in the communication bandwidth and latency among different combination of nodes and processors and the actual mapping between the 2D virtual processor grid and the physical layout of the processors.

Any network will have different levels of locality, and the realities of packaging dictate that the distance between computational nodes will vary and that subsets of nodes will share routers while other nodes will not. In most MPI implementations, ranks are assigned so that consecutive ranks first fill up a node, and then fill the closest node physically, and so on. It is thus very likely that jobs placed on machines ranks that are logically close in `MPI_COMM_WORLD` are also physically close to each other. Therefore the goal of our broadcast implementation should be to minimize the amount of data that needs to be transferred at long distance, both logically and physically, while at the same time avoiding hot spots in the network.

To reduce communication imbalance and consequently the communication cost, we modify the way the collective communication in step 3 of Algorithm 1 is implemented. Instead of using a "centralized" sender/receiver model for the broadcast and reduction in which the communication path can be described a `Flat-Tree` shown in Figure 3(a), we use a binary-tree based algorithm commonly employed in the implementation of `MPI_Bcast` and `MPI_Reduce`. Compared to the `Flat-Tree` model, which puts a heavy load on the root, the `Binary-Tree` based scheme reduces the total volume sent/received from the root from $p-1$ messages to two messages, and spreads the total communication volume among more processors. Figure 3(a) and 3(b)shows how messages are passed among different processors in a `Flat-Tree` and `Binary-Tree` based broadcast operation.

In order to implement a non-blocking `Binary-Tree` based collective communication scheme, destination processor of each message has to be specified in a hierarchical fashion. We will refer to processors that lie between the root and the leaves of the tree as "internal nodes". Such processors serve as the forwarding processors. The `Binary-Tree` is built by repeatedly splitting the ordered list of ranks in two partitions, and chose the first rank in each half to be the internal nodes at the current level.

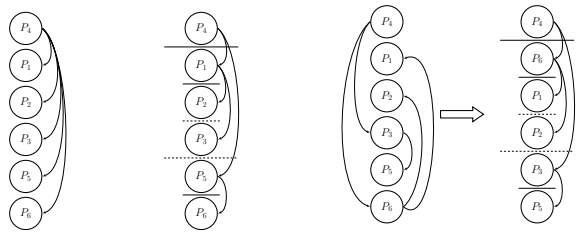
The `Binary-Tree` has two main benefits. First, the large reduction in the number of messages sent from the root greatly reduces the chances of an instantaneous hot spot in the network around the root node. Second, the `Binary-Tree` greatly increases the chance that data is exchanged between two ranks that are logically closer, and thus likely physically closer in the network, by putting them in the same partition. As an example, Figure 3(b) shows that the processors $P_1 - P_6$ are involved in a broadcast operation, with P_4 being the root. The `Flat-Tree` simply sends data from P_4 to all processors other than P_4. The `Binary-Tree` uses a pre-designed ordering, i.e. P_4 first sends to P_1 and P_5 which are of distance 1 from the root P_4. The data is further broadcast from P_1 to P_2, P_3, and from P_5 to P_6. The data communication can be performed recursively for deeper trees.

The drawback of the `Binary-Tree` is that due to the pipelining of different loop iterates in the outer-loop of Algorithm 1, each processor may participate in several non-blocking restricted collective communication simultaneously. If that processor is an internal node in several binary communication trees, the total volume from those many broadcasts passing through that processor can be much larger than that sent by other processors. One can see that with this scheme, the highest numbered rank in a column will never be chosen as an internal node and thus will never forward any data. On the other hand, the lowest numbered rank in a column will always be chosen as an internal node and thus will always forward data. While the exact ranks chosen as internal nodes will vary depending on the root, patterns of communication intensity will develop throughout the range of ranks. Such a striped pattern is clearly seen in the communication volume heat map seen in Figure 5(b).

To alleviate this problem, we use a heuristic method that involves applying a random circular shift to the list of receiving processor ranks. Such a procedure, referred to as `Shifted Binary-Tree`, is depicted in Figure 3(c). A random position is selected in the sorted list of ranks and a circular shift is then executed around this position. The random shift makes it therefore less likely that same processors will be picked as internal nodes when building multiple binary trees. The rationale of this circular shift is therefore to smooth the total communication load across all processors.

In our example in Figure 3(c), the `Shifted Binary-Tree` breaks the pre-designed and monotonically increasing ordering of ranks involved in the tree, and picks a random processor other than the root (P_4) to be the first child. The rest of the processors follow circularly, so that the sequence P_4,P_6,P_1,P_2,P_3,P_5 can be regarded as a re-ordered list to generate the `Binary-Tree`. This results in a different data communication pattern.

One can also consider using a fully random permutation of processor ranks. However, such a permutation would reduce network locality by putting ranks which are logically "closer" far from each other. Moreover, our experiments show that this approach leads to deteriorated load balancing in terms of communication volume compared to `Shifted Binary-Tree`.



(a) `Flat-Tree` (b) `Binary-Tree` (c) `Shifted Binary-Tree`

Figure 3: Various possible tree-based communication patterns for the broadcast operation.

Both these binary tree structures do an excellent job of reducing "hot spots" in the network and reducing the communication distance for data transfer. As already discussed, the simple `Binary-Tree` structure reduces the number of messages transferred to/from the root from $p - 1$ to just two, greatly reducing the chance of an instantaneous hot spot developing at the time of that broadcast. It also increases the chances that data will be transferred between ranks that are logically close to each other, rather than all data being transferred from the root to the leaf no matter the distance between the leaf and the root.

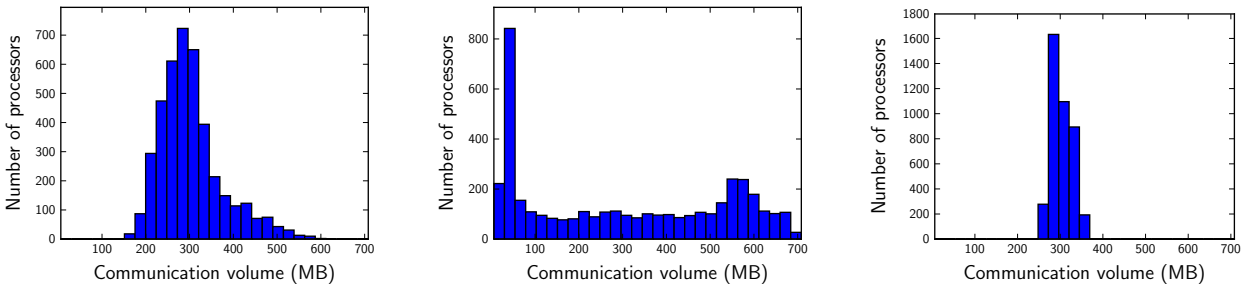
The `Shifted Binary-Tree` further reduces the chances of hot spots in the network given that there are many broadcasts happening simultaneously. The communication heat map in Figure 5(c) clearly shows that the communication volume spreads much more evenly across all ranks. The maximum amount of data sent by any rank is also reduced. Note that the circular shift potentially reduces network locality by putting the highest rank in the list before the lowest rank. However, this does not negatively impact the algorithm in any significant way, since the root and the next level of internal nodes were not guaranteed to be close to each other when the number of processors involved in a communication is relatively large.

4. NUMERICAL RESULTS

We now report the outcome of a number of computational experiments conducted to analyze the communication volume and pattern of `PSelInv`, and to evaluate and compare the efficiency of different ways to implement the restricted collective communication pattern required in `PSelInv`.

In all of our experiments, we used the NERSC Edison platform with Cray XC30 nodes. Each node has 24 cores partitioned among two Intel Ivy Bridge processors. Each 12-core processor runs at 2.4GHz. A single node has 64GB of memory, providing more than 2.6 GB of memory per core.

To evaluate the performance of `PSelInv`, we use two matrices of different sizes and sparsity patterns. The DG_PNF14000 matrix is generated from the electronic structure calculation of a 2D phosphorene nanoflake with $14,000$ atoms. The matrix is a discretized Kohn-Sham Hamiltonian obtained from an adaptive local basis expansion scheme combined with a discontinuous Galerkin framework [20]. This matrix is relatively dense. The matrix size is 512,000, with 0.2% nonzeros in A and 1.3% nonzeros in the L and U fac-



(a) `Flat-Tree` (b) `Binary-Tree` (c) `Shifted Binary-Tree`

Figure 4: Communication volume distribution of `Col-Bcast`

tors. The second matrix is named audikw_1 matrix obtained from the University of Florida matrix collection [21]. This matrix is relatively sparse. The matrix size is 943,695, with 0.009% nonzeros in A and 0.3% nonzeros in the L and U factors. These two matrices represent two different scenarios in terms of total communication volume. For the DG_PNF14000 matrix, the communication volume of is expected to be very large. This can lead to imbalanced data communication on different processors. For the audikw_1 matrix, the scalability of `PSelInv` is more limited by a larger communication over computation ratio.

4.1 Communication load analysis

The first set of experiments aims at analyzing the communication load among different processors, and comparing the efficiency of using different types of tree structures to implement restricted collective communications by using asynchronous MPI functions. We report the total data volume sent from each processor on a 64-by-64 $= 4,096$ processor grid for the audikw_1 matrix.

Our main focus is the broadcast of blocks of $\hat{L}^T_{\mathcal{I},\mathcal{K}}$ within each column group, and the reduction of $A^{-1}_{\mathcal{J},\mathcal{I}}\hat{L}_{\mathcal{I},\mathcal{K}}$ within each row group. These two operations are the most expensive communication steps of `PSelInv`, and we refer to them as `Col-Bcast` and `Row-Reduce` respectively. We report the minimum and maximum outgoing volume of data among all processors in Table 1 for different types of tree-based collective communication schemes.

In Figure 5(a), we report the volume sent during `Col-Bcast` using the `Flat-Tree` pattern, as used in the `PSelInv` developed in [17] (currently released under the PEXSI package v0.7.3, referred to as `PSelInv` v0.7.3 below). We observe that the data volume associated with processors near the diagonal of the 2D processor grid is significantly higher than those associated with off-diagonal processors. Significant variation of communication volume can also be seen among the diagonal processors themselves. Furthermore, from the distribution of the processor load shown in Figure 4(a), we observe that some processors send more than twice the average volume of data sent by all processors. This load imbalance creates contention on the network and limits the strong scalability of `PSelInv`.

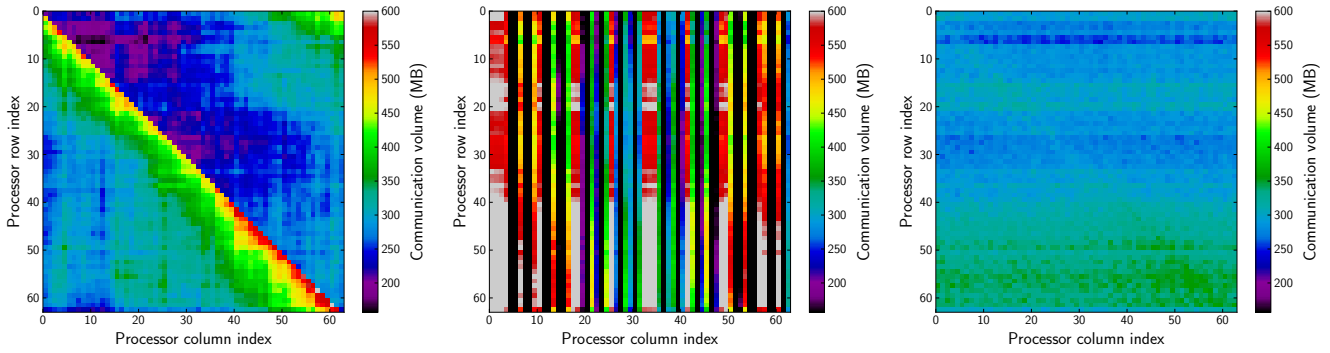
When a simple `Binary-Tree` is used to organize the way messages are sent from the root to other processors involved in the restricted collective communication, load imbalance can still be observed from the heat map given in Figure 5(b). It can be seen from Table 1 and Figure 4(b) that the maximum communication volume among all processors and the standard deviation are actually higher than those in the `Flat-Tree` based communication. Although most of the nodes see their load decreased and the median communication volume reduced from 291 MB to 288 MB, ranks in the last quartile of the most loaded processors send more than 536 MB of data instead of the 331 MB of data sent using a `Flat-Tree` based scheme. This observation confirms that some processors participate in multiple `Binary-Tree` based broadcasts as internal nodes.

Communication tree	Min	Max	Median	Std. dev
`Flat-Tree`	156.951	600.168	291.595	72.048
`Binary-Tree`	5.89874	708.268	288.851	226.565
`Shifted Binary-Tree`	238.647	363.336	298.58	24.0957

Table 1: Volume sent during `Col-Bcast` (in MB) for the audikw_1 matrix.

In order to reduce the likelihood of a processor being chosen repeatedly as an internal node, we introduced the `Shifted Binary-Tree` communication scheme in Section 3. The communication volume heat map associated with this scheme is shown in Figure 5(c). We use the same colorbar that we used for Figure 5(a) so that we can compare these two heat maps directly. We can clearly see that the overall heat map is much "cooler". The communication "hot spots" appear to be eliminated by the `Shifted Binary-Tree`. As we explained in section 3, the reduction in the overall communication load and the removal of the "hot spots" result from shifting processor ranks in such a way that different processors are picked as internal nodes of different communication trees. The effect of this is clearly observed in practice on the minimum and maximum volumes, given in Table 1. The variation of the communication volume among different processors is significantly reduced (resp. MIN 238 MB and MAX 363 MB) than that using `Flat-Tree` (resp. MIN 156 MB and MAX 600 MB). The standard deviation is significantly reduced from 72 MB to 24 MB, confirming the efficiency of the approach.

When fewer processors are used to perform `PSelInv`, the communication load imbalance might not be so severe. Fig-



(a) `Flat-Tree` (b) `Binary-Tree` (c) `Shifted Binary-Tree`

Figure 5: Communication volume heat map of `Col-Bcast`.

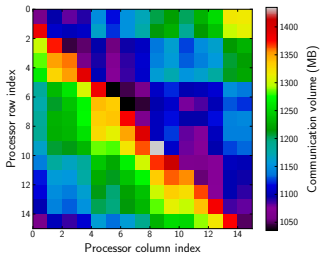
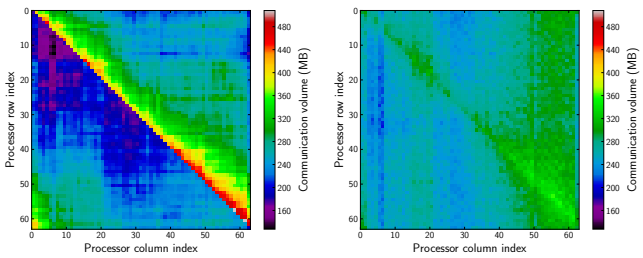


Figure 6: Communication volume heat map of `Col-Bcast` using `Flat-Tree` on 256 processors

ure 6 depicts the communication volume heat map of `Col-Bcast` for the same audikw_1 matrix running on a 16-by-16 processor grid using the `Flat-Tree` scheme. In this case, the average volume is 1185.77 MB, while the standard deviation is 96.02 MB, corresponding to 8% of the average. This is sharply lower than the 23.7% standard deviation when `PSelInv` is carried out on 4,096 processors.



(a) `Flat-Tree` (b) `Shifted Binary-Tree`

Figure 7: Communication volume heat map of `Row-Reduce`

The `Row-Reduce` operation can be seen as the reverse operation of a broadcast. In this case, it is the amount of data *received* by each processor that we are concerned with. Heat maps corresponding to `Flat-Tree` and `Shifted Binary-Tree` are shown in Figures 7(a) and 7(b) respectively. We can clearly see that the `Shifted Binary-Tree` scheme results in a more balanced communication load distribution among all processors.

Altogether, our experimental results demonstrate that the use of a binary tree to organize messages in a restricted collective communication does mitigates the inherent load imbalance of the `Flat-Tree` communication pattern. However, since multiple restricted collective communications may take place at the same time with some of the processors participating in all of them, the binary trees associated with these collective calls have to be built in such a way that these processors are not always picked as internal nodes of the binary trees. This can be achieved using the proposed `Shifted Binary-Tree` communication pattern.

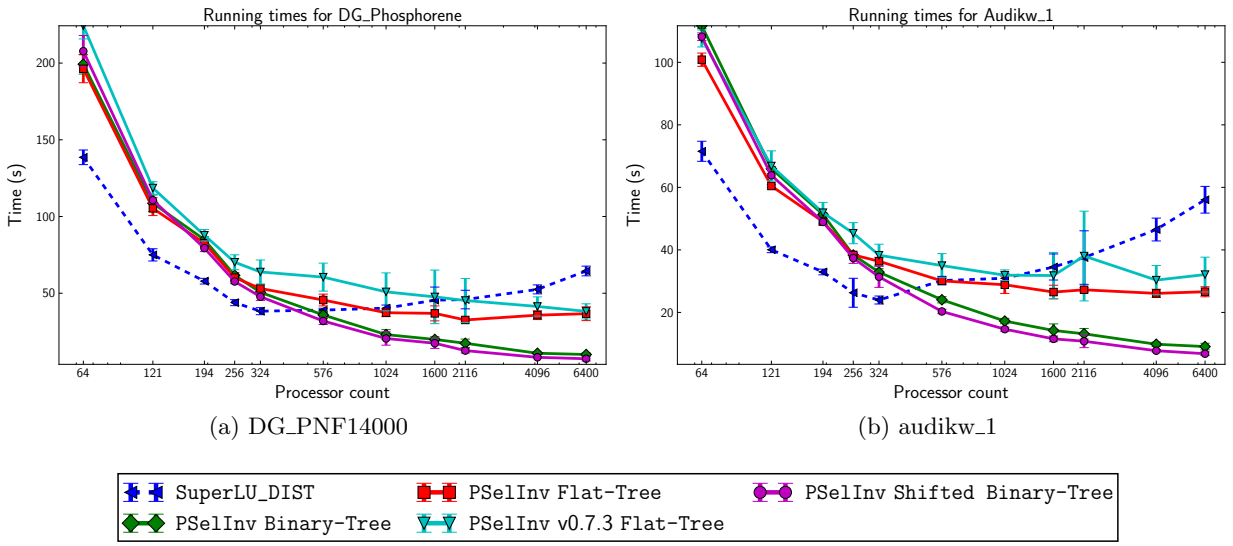
4.2 Impact on performance

In this section, we assess the impact of using different tree-based restricted collective communication schemes on the overall performance of `PSelInv`, and compare the strong scaling of `PSelInv` using either (1) `Flat-Tree`, (2) `Binary-Tree` and (3) `Shifted Binary-Tree` communication patterns.

The new implementation of `PSelInv` contains additional code improvements that do not fall into the scope of this paper. Therefore, to emphasize the impact of the different implementations of restricted collective communication only, we use the wallclock timing measurements of the new `PSelInv` using the `Flat-Tree` approach as the baseline for comparison. We also provide timings from our previous v0.7.3 release of `PSelInv` [17] for reference. This implementation also uses a `Flat-Tree` communication pattern. The wallclock time of the *LU* factorization based on `SuperLU_DIST` [18] is also provided. This is a pre-processing step of `PSelInv`. The `SuperLU_DIST` timing results are also used here as a reference for evaluating the strong scaling `PSelInv`.

Every data point generated from the strong scaling experiments presented in this section corresponds to the average of 6 runs. We report standard deviations by using error bars. We define the speedup factor and other ratios as the ratio between average values.

Results for the DG_PNF14000 matrix are depicted in Figure 8(a). We observe that switching from the `Flat-Tree` to the `Binary-Tree` scheme leads to a reduction of the wall



(a) DG_PNF14000 (b) audikw_1

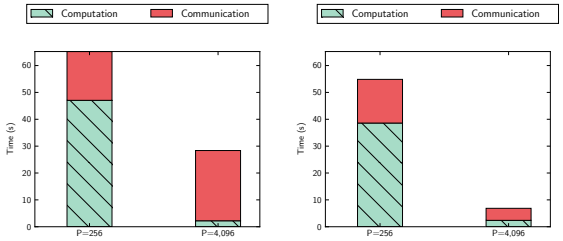
Figure 8: Running times of `PSelInv` for two sample matrices

clock time by a factor of 1.7 on average. The reduction factor is larger when a larger number of processors are used in `PSelInv`. In particular, when more than 1,024 processors are used, the average speedup factor is 2.5. The speedup factor reaches 3.6 when the computation is performed on 6,400 processors. Additional performance improvement can be seen when `Shifted Binary-Tree` scheme is used. In particular, the average speedup factor is increased to 2.0. When more than 1,024 processors are used, the average speedup factor increases to 3.2. The maximum speedup reaches 5.0x when 6,400 processors are used.

Switching from `Flat-Tree` to `Binary-Tree` also reduces the standard deviation of the wall clock time among multiple runs of the same code on the same input by a factor of 1.35 on average. The reduction factor is 1.26 when the `Shifted Binary-Tree` is used. Compared to v0.7.3 of `PSelInv` presented in [17], the average reduction in standard deviation resulting from the use of `Shifted Binary-Tree` is more than 3.19.

Similar observations holds for the audikw_1 matrix. The strong scaling plot depicted in Figure 8(b) demonstrates the use of `Binary-Tree` and `Shifted Binary-Tree` allows us to scale the computation to 6,400 processors, whereas the scalability of `Flat-Tree` based `PSelInv` calculations is limited to less than 1,024 processors. The standard deviation in running time is reduced by more than a factor of 4 when a large number of processors are used to run the same program with the same input multiple times.

The improved scalability of `PSelInv` clearly results from a better implementation of restricted collective communication which significant reduces communication overhead. This can also be seen from the ratio of computation and communication time. In Figures 9(a) and 9(b), we plot both the communication and computation time consumed by `PSelInv` for the DG_PNF14000 matrix when the computation is carried out on 256 and 4,096 processors respectively. The communication to computation ratio is reduced from 11.8 to 1.9 when we switch from a `Flat-Tree` based communication scheme a `Shifted Binary-Tree` based scheme.



(a) `Flat-Tree` (b) `Shifted Binary-Tree`

Figure 9: Computation and communication times for DG_PNF14000

It is interesting to note that in both test cases, the benefit of using `Shifted Binary-Tree` is not so pronounced when the `PSelInv` is carried out among a small set of processors (e.g. 256). This is due to the fact that in this case, several restricted collective communications take place within a single node of Edison, which has 24 cores. Because message passing is implemented as memory copies within shared memory on a single node, its cost is generally lower compared to internode communication. Moreover, having a single send buffer in a `Flat-Tree` based scheme could enhance cache reuse and reduces the impact of issuing p messages compared to the $\log_2 p$ messages sent by a binary tree based collective communication scheme. Therefore, in practice, one can potentially use a "hybrid" scheme in which a `Flat-Tree` based collective communication is used when the communication is restricted to a relatively small number of processors and a `Shifted Binary-Tree` based scheme is used when a large

number of processors are involved.

5. CONCLUSION AND FUTURE WORK

We described several implementations of restricted collective communication in a parallel selected inversion algorithm. Each implementation uses the point-to-point `MPI_Isend` and `MPI_Irecv` functions available in a standard MPI library. However, they differ in the way each message is moved from one processor to another. We showed that a binary tree based data propagation scheme is far superior than a flat tree based scheme when a large number of processors are involved in the collective communication. In particular, the binary-tree based scheme minimizes communication load imbalance and removes communication "hot spots".

The use of `MPI_Isend` and `MPI_Irecv` allows multiple collective communications to be initiated at the same time. This is a desired feature that would allow us to exploit a higher level of concurrency in the `PSelInv` algorithm. In order to prevent a processor from becoming an internal node of multiple binary trees, we developed a heuristic that involves applying a random circular shift to the list of receiving processor ranks. We demonstrated that such a heuristic leads to a significant improvement in the scalability of `PSelInv`. For instance, when $6,400$ processors are used, we observe over 5x speedup for test matrices. It also reduces the variation in running time when the same program is executed multiple times with the same input. Such variation is caused by inhomogeneous network architecture. Reducing this type of variation is extremely important for achieving scalable performance of the PEXSI algorithm [2, 3, 5] in which multiple selected inversions are carried out simultaneously on different subgroups of processors. Although our implementation in this work is for symmetric matrices, the same communication strategy can be naturally extended to asymmetric matrices, which is our work in progress.

Acknowledgment

This work was partially supported by the Scientific Discovery through Advanced Computing (SciDAC) program (M. J., L. L. and C. Y.), and the Center for Applied Mathematics for Energy Research Applications (CAMERA) (L. L. and C. Y.), which are partnerships between Basic Energy Sciences (BES) and Advanced Scientific Computing Research (ASCR) at the U.S Department of Energy.

6. REFERENCES

[1] L. Lin, J. Lu, L. Ying, and W. E. Pole-based approximation of the Fermi-Dirac function. *Chin. Ann. Math.*, 30B:729, 2009.

[2] L. Lin, J. Lu, L. Ying, R. Car, and W. E. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. *Comm. Math. Sci.*, 7:755, 2009.

[3] L. Lin, C. Yang, J. Lu, L. Ying, and W. E. A fast parallel algorithm for selected inversion of structured sparse matrices with application to 2D electronic structure calculations. *SIAM J. Sci. Comput.*, 33:1329, 2011.

[4] L. Lin, C. Yang, J. Meza, J. Lu, L. Ying, and W. E. SelInv – An algorithm for selected inversion of a sparse symmetric matrix. *ACM. Trans. Math. Software*, 37:40, 2011.

[5] L. Lin, M. Chen, C. Yang, and L. He. Accelerating atomic orbital-based electronic structure calculation via pole expansion and selected inversion. *J. Phys. Condens. Matter*, 25:295501, 2013.

[6] W. Kohn and L. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, 1965.

[7] K. Takahashi, J. Fagan, and M. Chin. Formation of a sparse bus impedance matrix and its application to short circuit study. In *8th PICA Conf. Proc.*, 1973.

[8] A. Erisman and W. Tinney. On computing certain elements of the inverse of a sparse matrix. *Comm. ACM*, 18:177, 1975.

[9] Y. E. Campbell and T. A. Davis. Computing the sparse inverse subset: an inverse multifrontal approach. Technical Report TR-95-021, University of Florida, 1995.

[10] S. Li, S. Ahmed, G. Klimeck, and E. Darve. Computing entries of the inverse of a sparse matrix using the FIND algorithm. *J. Comput. Phys.*, 227:9408–9427, 2008.

[11] S. Li and E. Darve. Extension and optimization of the find algorithm: Computing Green's and less-than Green's functions. *J. Comput. Phys.*, 231:1121–1139, 2012.

[12] S. Li, W. Wu, and E. Darve. A fast algorithm for sparse matrix computations related to inversion. *J. Comput. Phys.*, 242:915–945, 2013.

[13] U. Hetmaniuk, Y. Zhao, and M. P. Anantram. A nested dissection approach to modeling transport in nanodevices: Algorithms and applications. *Int. J. Numer. Meth. Eng.*, 2013.

[14] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. *SIAM J. Sci. Comput.*, 34:A1975–A1999, 2012.

[15] P. R. Amestoy, I. S Duff, J. Y. L'Excellent, and F. H. Rouet. Parallel computation of entries of A^{-1} . Technical report, CERFACS, Toulouse, France, 2012.

[16] D. E. Petersen, S. Li, K. Stokbro, H. H. B. Sørensen, P. C. Hansen, S. Skelboe, and E. Darve. A hybrid method for the parallel computation of Green's functions. *J. Comput. Phys.*, 228:5020–5039, 2009.

[17] M. Jacquelin, L. Lin, and C. Yang. PSelInv–a distributed memory parallel algorithm for selected inversion: the symmetric case. *arXiv:1404.0447*, 2014.

[18] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Software*, 29:110, 2003.

[19] J. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11:134, 1990.

[20] L. Lin, J. Lu, L. Ying, and W. E. Adaptive local basis set for Kohn-Sham density functional theory in a discontinuous Galerkin framework I: Total energy calculation. *J. Comput. Phys.*, 231:2140–2154, 2012.

[21] T. A. Davis and Y. Hu. The University of Florida

sparse matrix collection. *ACM Trans. Math. Software*, 38:1, 2011.